11-1998

Preliminary Design of JML: A Behavioral Interface Specification

Gary T. Leavens
Iowa State University

Albert L. Baker
Iowa State University

Clyde Ruby
Iowa State University

Preliminary Design of JML: A Behavioral Interface Specification

Abstract
JML is a behavioral interface specification language tailored to Java. It also allows assertions to be intermixed with Java code, as an aid to verification and debugging. JML is designed to be used by working software engineers, and requires only modest mathematical training. To achieve this goal, JML uses Eiffel-style assertion syntax combined with the model-based approach to specifications typified by VDM and Larch. However, JML supports quantifiers, specification-only variables, frame conditions, and other enhancements that make it more expressive for specification than Eiffel. This paper discusses the goals of JML, the overall approach, and describes the language through examples. It is intended for readers who have some familiarity with both Java and behavioral specification using pre- and postconditions.

Keywords
Behavioral interface specification, Java, JML, Eiffel, Larch, model-based specification, precondition, postcondition, frame

Disciplines
Programming Languages and Compilers | Systems Architecture

Preliminary Design of JML: A Behavioral Interface Specification Language for Java

Gary T. Leavens, Albert L. Baker and Clyde Ruby

TR #98-06b
June 1998, revised July, November 1998

Keywords: Behavioral interface specification, Java, JML, Eiffel, Larch, model-based specification, precondition, postcondition, frame.

1996 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, theory, Larch, Eiffel, JML; D.2.7 [*Software Engineering*] Distribution and Maintenance — documentation; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, logics of programs, pre- and post-conditions, specification techniques.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Preliminary Design of JML: A Behavioral Interface Specification Language for Java

Gary T. Leavens*, Albert L. Baker, and Clyde Ruby
Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavens@cs.iastate.edu, baker@cs.istate.edu, ruby@cs.iastate.edu

November 4, 1998

Abstract

JML is a behavioral interface specification language tailored to Java. It also allows assertions to be intermixed with Java code, as an aid to verification and debugging. JML is designed to be used by working software engineers, and requires only modest mathematical training. To achieve this goal, JML uses Eiffel-style assertion syntax combined with the model-based approach to specifications typified by VDM and Larch. However, JML supports quantifiers, specification-only variables, frame conditions, and other enhancements that make it more expressive for specification than Eiffel.

This paper discusses the goals of JML, the overall approach, and describes the language through examples. It is intended for readers who have some familiarity with both Java and behavioral specification using pre- and postconditions.

1 Introduction

JML stands for "Java Modeling Language." JML is a *behavioral interface specification language* (BISL) [42] designed to specify Java [1, 7] modules. Java *modules* are classes and interfaces.

The main goal of the research presented in this paper is to better understand how to make BISLs (and BISL tools) that are practical and effective for production software environments. In order to understand this goal, and the more detailed discussion of our goals for JML, it helps to define more precisely what a behavioral interface specification is. After doing this, we return to describing the goals of JML, and then give an outline of the rest of the paper.

1.1 Behavioral Interface Specification

As a BISL tailored to the specification of Java modules, JML describes two important aspects of a Java module:

- its *interface*, which consists of the names and static information found in Java declarations, and

*The work of Leavens and Ruby is supported in part by a grant from Rockwell International Corporation and by NSF grant CCR-9503168. The work of Leavens and Baker is supported in part by the NSF grant CCR 9803843.

```
public class IntMathOps {                                          // 1
  public static int isqrt(int y)                                   // 2
       //@ behavior {                                              // 3
       //@   requires y >= 0;                                      // 4
       //@   ensures result * result <= y                          // 5
       //@           && y < (result + 1) * (result + 1);           // 6
       //@ }                                                       // 7
  { return (int) java.lang.Math.sqrt(y); }                         // 8
}                                                                  // 9
```

Figure 1: A JML specification written as annotations in the Java code file `IntMathOps.java`.

- its *behavior*, which tells how the module acts when used.

Because they describe interface details for clients written in a specific programming language, BISLs are inherently language-specific [42]. For example, a BISL tailored to C++, such as Larch/C++ [15], describes how to use a module in a C++ program. A Larch/C++ specification cannot be implemented correctly in Java, and a JML specification cannot be correctly implemented in C++ (except for functions that are specified as native code).

JML specifications are designed to be annotations in Java code files [26, 37, 38]. To a Java compiler such annotations are comments that are ignored. This allows JML specifications, such as the specification in Figure 1, to be embedded in Java code files. It is possible, however, to have specifications that are separate from code, if desired; this can be done as in Figure 2. Most of our examples, however, will be Java code files, as we expect most users to use this form. Whatever users prefer, we expect that for each module they would only use one of these formats, not both.

As a simple example of a behavioral interface specification in JML, consider the specification in Figure 1. This figure specifies a Java class, `IntMathOps` that contains one static method (function member) named `isqrt`. The single-line comments to the far right (which start with `//`) give the line numbers in this specification; they are ignored by both Java and JML. Comments with an immediately following at-sign (`@`), as on lines 3–7, are *annotations*, which are treated as comments by a Java compiler, but the text following the annotation marker is meaningful in JML.

In Figure 1, interface information is declared in lines 1 and 2. Line 1 declares a class named `IntMathOps`, and line 2 declares a method named `isqrt`. Note that all of Java's declaration syntax is allowed in JML, including, on lines 1 and 2, that the names declared are `public`, that the method is `static` (line 2), that its return type is `int` (line 2), and that it takes one `int` argument.

Such interface declarations must be found in a Java module that correctly implements this specification. This is automatically the case in Figure 1, since that file also contains the implementation. In fact, when Java annotations are embedded in "`.java`" files, the interface specification is the actual Java source code. To be correct, an implementation must have both the specified interface and the specified behavior.

In Figure 1, the behavioral information is specified in the annotations on lines 3–7. The behavioral part of a specification is found between an opening `behavior {` (line 3) and a

```
public class IntMathOps {
  public static int isqrt(int y);
    behavior {
      requires y >= 0;
      ensures result * result <= y
              && y < (result + 1) * (result + 1);
  }
}
```

Figure 2: A form of the previous specification is contained in a `.jml` file, `IntMathOps.jml`. This form allows the code for concrete methods to be omitted. Note that a semicolon comes before the `behavior` keyword when the code is omitted.

closing `}` (line 7), ignoring the annotation markers (`//@`). The keyword `behavior` is used to make the following specification distinct (in the syntax) from the code block that follows it. Between these is a precondition, which follows the keyword `requires` on line 4, and a postcondition, which follows the keyword `ensures` on line 5. The precondition says what must be true about the arguments (and other parts of the state); if the precondition is true, then the method must terminate in a state that satisfies the postcondition. This is a contract between the caller of the method and the implementor [10, 27]. The caller is obligated to make the precondition true, and gets the benefit of having the postcondition then be satisfied. The implementor gets the benefit of being able to assume the precondition, and is obligated to make the postcondition true in that case.

In general, pre- and postconditions in JML are written using an extended form of Java expressions. In this case, the only extension visible is the keyword `result`, which is used in the postcondition to denote the value returned by the method. The type of `result` is the return type of the method; for example, the type of `result` in `isqrt` is `int`. The postcondition says that the result is an integer approximation to the square root of `y`. Note that the behavioral specification does not give an algorithm for finding the square root.

As shown in Figure 1, JML can add annotations directly to classes containing Java code. But one can also use JML to write documentation in separate non-Java "`.jml`" files. Since these files are not Java code files, JML allows the user to omit the code for concrete methods in a class. Figure 2 shows how this is done, replacing the code by a semicolon (`;`), as in a Java abstract method declaration.

To summarize, a behavioral interface specification describes both the interface details of a module, and its behavior. The interface details are written in the syntax of the programming language; thus JML uses the Java declaration syntax. The behavioral specification uses pre- and postconditions.

1.2 Goals

As mentioned above, the main goal of our research is to better understand how to develop BISLs (and BISL tools) that are practical and effective. We are concerned with both technical requirements and with other factors such as training and documentation, although in the rest of this paper we will only be concerned with technical requirements for the BISL

itself. The practicality and effectiveness of JML will be judged by how well it can document reusable class libraries, frameworks, and Application Programming Interfaces (APIs).

We believe that to meet the overall goal of practical and effective behavioral interface specification, JML must meet the following subsidiary goals.

- JML must be able to document the interfaces and behavior of existing software, regardless of the analysis and design methods used to create it.

  If JML were limited to only handling certain Java features or certain kinds of software, then some APIs would not be amenable to documentation using JML. Since the effort put into writing such documentation will have a proportionally larger payoff for software that is more widely reused, it is important to be able to document existing reusable software components. This is especially true since software that is implemented and debugged is more likely to be reused than software that has yet to be implemented.

- The notation used in JML should be readily understandable by Java programmers, including those with only standard mathematical training.

  A preliminary study by Finney [5] indicates that graphic mathematical notations, such as those found in Z [9, 35] may make such specifications hard to read, even for programmers trained in the notation. This accords with our experience in teaching formal specification notations to programmers. Hence, our strategy for meeting this goal has been to shun most special-purpose mathematical notations in favor of Java's own expression syntax.

- The language must be capable of being given a rigorous, formal semantics, and must also be amenable to tool support.

  This goal also helps ensure that the specification language does not suffer from logical problems, which would make it less useful for static analysis, prototyping, and testing tools.

We also have in mind a long range goal of a specification compiler, that would produce prototypes from constructive specifications [39].

As a general strategy for achieving these goals, we have tried to blend the Eiffel [27, 28, 29] and Larch [42, 43, 8, 16] approaches to specification. From Eiffel we have taken the idea that assertions can be written in a language that is based on Java expressions. We also use the `old` notation from Eiffel, as described below, instead of the Larch style annotation of names with state functions. However, Eiffel specifications, as written by Meyer, are typically not as detailed as model-based specifications written, for example, in Larch BISLs or VDM [12]. Hence, we have combined these approaches, by using syntactic ideas from Eiffel and semantic ideas from model-based specification languages.

JML also has some other differences from Eiffel (and its cousins Sather and Sather-K). The most important is the concept of specification-only declarations. These declarations, as will be explained below, allow more abstract and exact specifications of behavior than is typically done in Eiffel; they allow one to write specifications that are similar to the spirit of VDM or Larch BISLs. A major difference is that we have extended the syntax of Java expressions with quantifiers and other constructs that are needed for logical expressiveness, but which are not always executable. Finally, we ban side-effects and other problematic features of code in assertions.

On the other hand, our experience with Larch/C++ has taught us to adapt the model-based approach in two ways, with the aim of making it more practical and easy to learn. The first adaptation is again the use of specification-only model (or ghost) variables. An object will thus have (in general) several such *model fields*, which are used only for the purpose of describing, abstractly, the values of objects. This simplifies the use of JML, as compared with most Larch BISLs, since specifiers (and their readers) hardly ever need to know about algebraic style specification. It also makes designing a model for a Java class or interface similar, in some respects, to designing an implementation data structure in Java. We hope that this similarity will make the specification language easier to understand. (This kind of model also has some technical advantages that will be described below.)

The second adaptation is the hiding of the details of mathematical modeling are hidden behind a facade of Java classes. In the Larch approach to behavioral interface specification [42], the mathematical notation used in assertions is presented directly to the specifier. This allows the same mathematical notation to be used in many different specification languages. However, it also means that the user of such a specification language has to learn a notation for assertions that is different than their programming language's notation for expressions. In JML we use a compromise approach, hiding these details behind Java classes. These classes are pure, in the sense that they reflect the underlying mathematics, and hence do not use side-effects (at least not in any observable way). Besides insulating the user of JML from the details of the mathematical notation, this compromise approach also insulates the design of JML from the details of the mathematical logic used for theorem proving.

1.3 Outline

In the next sections we describe more about JML and its semantics. Section 2 uses examples to show how Java classes and interfaces are specified; this section also briefly describes the semantics of subtyping and refinement. Section 3 describes more detail about the expressions that can be used in predicates. Section 4 presents conclusions from our preliminary design effort. Finally, the appendix gives the syntax of JML.

2 Class and Interface Specifications

In this section we give some examples of JML class specifications that illustrate the features of JML.

2.1 Abstract Models

A simple example of an abstract class specification is the ever-popular `UnboundedStack` type, which is presented in Figure 3. This figure has the abstract values of stack objects specified by the model data field `theStack`, which is declared on the fourth non-blank line. Since it is declared using the modifier `model`, such a field does not have to be implemented; however, for purposes of the specification we treat it exactly as any other Java field (i.e., as a variable). That is, we imagine that each instance of the class `UnboundedStack` has such a field.

The type of the model field `theStack` is a pure type, `JMLObjectSequence`, which is a sequence of objects. It is provided by JML in the package `edu.iastate.cs.jml.models`, which is imported in the second non-blank line of the figure.[1] Note that this `import`

[1] Users can also define their own pure types, as we will show below.

```
package edu.cs.iastate.jml.samples.stacks;

//@ model import edu.cs.iastate.jml.models.*;

public abstract class UnboundedStack {

  //@ public model JMLObjectSequence theStack;

  //@ public initially theStack.isEmpty();

  public abstract void pop( );
    //@ behavior {
    //@   requires !theStack.isEmpty();
    //@   modifiable theStack;
    //@   ensures theStack.equals(old(theStack.trailer()));
    //@ }

  public abstract void push(Object x);
    //@ behavior {
    //@   modifiable theStack;
    //@   ensures theStack.equals(old(theStack.addFirst(x)));
    //@ }

  public abstract Object top( );
    //@ behavior {
    //@   requires !theStack.isEmpty();
    //@   ensures result == theStack.first();
    //@ }
}
```

Figure 3: A specification of the abstract class `UnboundedStack` (file `UnboundedStack.java`).

declaration does not have to appear in the implementation, since it is modified by the keyword `model`. In general, any declaration form in Java can have this modifier, with the same meaning: that the declaration in question is only used for specification purposes, and does not have to appear in an implementation.

Following the declaration of the model field, above the specification of `pop` in Figure 3, is an `initially` clause. (Such clauses are adapted from Resolve [31].) This clause is declared `public`, since it only refers to public model fields.

An `initially` clause permits data type induction ([11, 44]) for abstract classes and interfaces, by supplying a property that must appear to be true of the starting states of objects. In each visible state (outside of the methods of `UnboundedStack`) all reachable objects of the type `UnboundedStack` must have a value that makes them appear to have been created as empty stacks and subsequently modified using the type's methods.

Following the `initially` clauses are the expected specifications of the `pop`, `push`, and `top` methods.

The use of the `modifiable` clauses in the behavioral specifications of `pop` and `push` is interesting (and another difference from Eiffel). These give frame conditions [2], which say that no objects, other than those mentioned (and those on which these objects depend, as explained below) may have their values changed.[2] When the `modifiable` clause is omitted, as it is in the specification of `top`, this means that no objects can have their state modified by the method's execution. Our interpretation of this is very strict, as even benevolent side effects are disallowed if the `modifiable` clause is omitted [21, 20].

When a method can modify some objects, these objects have different values in the pre-state and post-state of that method. Often the post-condition must refer to both of them. This is a notation similar to Eiffel's is used, to refer to the pre-state value of a variable. In JML the syntax is `old(`E`)`.[3] The meaning of `old(`E`)` is as if E were evaluated in the pre-state and that value is used in place of `old(`E`)` in the assertion. If E denotes an object that is modifiable, then the expression may not mean what is desired. Hence it is best if E is disjoint from any objects mentioned in the `modifiable` clause. This is automatically satisfied if E denotes a primitive value (such as an `int`). If that is not possible, then it is often safe if the type of E is a pure type.

For example, in `pop`'s postcondition the expression `old(theStack.trailer())` has type `JMLObjectSequence`, which is a pure type. The value of `theStack.trailer()` is computed in the *pre-state* of the method (just after the method is called and parameters have been passed, but before execution of the body).

Note also that, since `JMLObjectSequence` is a reference type, one is required to use `equals` instead of `==` to compare them for equality of values. (Using `==` would be a mistake, since it would only compare them for object identity, which in combination with `new` would always yield false.)

The specification of `push` does not have a `requires` clause. This means that the method imposes no obligations on the caller. (Logically, the meaning of an omitted `requires` clause is that the method's precondition is `true`, which is satisfied by all states, and hence imposes no obligations on the caller.) This seems to imply that the implementation must provide a literally unbounded stack, which is surely impossible. We avoid this problem, by

[2] An object is modified by a method when it is allocated in both the pre- and post-states of the method, and when some of its variables (model or concrete) change their values. This means that allocating objects, using Java's `new` operator, does not cause a modification.

[3] We use explicit parentheses following `old`, which indicates the expression to be evaluated in the pre-state explicitly; this is a difference from Eiffel.

following Poetzsch-Heffter [33] in releasing implementations from their obligations to fulfill the postcondition when Java runs out of storage. That is, a method implementation is correct if, whenever it is called in a state that satisfies its precondition, either

- the method terminates in a state that satisfies its postcondition, having modified only the objects permitted by its `modifiable` clause, or
- Java signals an error, by throwing an exception that inherits from `Error`.

2.2 Dependencies, Representations, and Exceptions

In this subsection we describe how model fields can be related to one another, and how dependencies among them affect the meaning of the `modifiable` clause. For this purpose we give two specifications, `BoundedThing` and `BoundedStack`. Along the way we also demonstrate how to specify methods that can throw exceptions and other features of JML.

Figure 4 is an interface specification with a simple abstract model. In this case, there are two model fields `MAX_SIZE` and `size`. The variable `MAX_SIZE` is a static model field, which is treated as a class variable, while `size` is a normal model field, and is thus treated as an instance variable.[4] In specifications of interfaces that extend or classes that implement this interface, these model fields are inherited. Thus, for example, every object that has a type that is a subtype of the `BoundedThing` interface is thought of, abstractly, as having a field `size`, of type `int`. Similarly, every class that inherits from `BoundedThing` is thought of as having a static model field `MAX_SIZE`.

Two pieces of class-level specification come after the abstract model in Figure 4.

The first is an `invariant` clause. An invariant does not have to hold during the execution of an object's methods, but it must hold, for each reachable object in each *visible state*; i.e., for each state outside of a public method's execution, and at the beginning and end of each such execution. The figure's invariant says that in every visible state, the `MAX_SIZE` variable has to be positive, and that every reachable object that is a `BoundedThing` must have a size field that has a value less than or equal to `MAX_SIZE`.

Following the invariant is a history constraint [23]. A history constraint is used to say how values can change between earlier and later states, such as a method's pre-state and its post-state. This prohibits subtypes from making certain state changes, even if they implement more methods than are specified in a given class. The history constraint in Figure 4 says that the value of `MAX_SIZE` cannot change, since in every pre-state and post-state (before and after the invocation of a method), its value in the post-state, written `MAX_SIZE`, must equal its value in the pre-state, written `old(MAX_SIZE)`.

Following the history constraint are the interfaces and specifications for four public methods.

The specification of the last method, `clone`, is somewhat interesting. Since `clone` may throw an exception, we use logical implication (written `=>`) and the JML primitive `returns` to say that, when the method returns without throwing an exception, then the result will be a `BoundedThing` and its size will be the same as the model field `size`. Note the use of the cast in the postcondition of `clone`, which is necessary, since the type of `result` is `Object`. (This also adheres to our goal of using Java syntax and semantics to the extent possible.) Note also that the conjunct `result instanceof BoundedThing` "protects" the next conjunct [18] since if it is false the meaning of the cast does not matter.

[4] Java does not allow fields to be declared in interfaces, but JML allows model fields in interfaces, since these are essential for defining the abstract values of the objects being specified.

```
package edu.iastate.cs.jml.samples.stacks;

public interface BoundedThing {

  //@ public model static int MAX_SIZE;
  //@ public model int size;

  //@ public invariant MAX_SIZE > 0 && 0 <= size && size <= MAX_SIZE;

  //@ public constraint MAX_SIZE == old(MAX_SIZE);

  public int getSizeLimit();
    //@ behavior {
    //@   ensures result == MAX_SIZE;
    //@ }

  public boolean isEmpty( );
    //@ behavior {
    //@   ensures result == (size == 0);
    //@ }

  public boolean isFull();
    //@ behavior {
    //@   ensures result == (size == MAX_SIZE);
    //@ }

  public Object clone () throws CloneNotSupportedException;
    //@ behavior {
    //@   ensures returns => result instanceof BoundedThing
    //@                     && size == ((BoundedThing)result.size);
    //@ }
}
```

Figure 4: A JML specification of an interface to bounded collection objects (file `BoundedThing.java`).

Finally, note that the use of `==` in this Figure 4 is okay, since in each case, the things being compared are primitive values, not references.

Figure 5 gives an interface for bounded stacks that extends the interface in Figure 4. In this specification, one can refer to `MAX_SIZE` from the `BoundedThing` interface, and to `size` as an inherited model field of objects.

The abstract model for `BoundedStackInterface` adds to the inherited model by declaring a model field named `theStack`. This field is typed as a `JMLObjectSequence`.

The `depends` and `represents` clauses that follow the declaration of `theStack` are an important feature in modeling with layers of model fields. They also play a crucial role in relating model fields to the concrete fields of objects, which can be considered to be the final layer of detail in a design. The `depends` clause says that `size` might change its value when the `theStack` changes, and the `represents` clause says how they are related. The `represents` clause gives additional facts that can be used in reasoning about the specification; in essence it tells how to extract the value of `size` from the value of `theStack`.[5] It serves the same purpose as an abstraction function in various proof methods for abstract data types (such as [11]).

The `invariant` that follows the `represents` clause in Figure 5 is our first example of checkable redundancy in a specification [17, 37, 38]. This concept is signaled in JML by the use of the keyword `redundantly`. It says both that the stated property is specified to hold and that this property is believed to follow from the other properties of the specification. In this case the invariant follows from the invariant inherited from the specification of `BoundedThing` and the fact stated in the `represents` clause. Even though this invariant is redundant, it is sometimes helpful to state such properties, since they are then brought to the attention of the readers of the specification.

Checking that such claimed redundancies really do follow from other information is also a good way to make sure that what is being specified is really what is intended. Such checks could be done manually, during reviews, or by an automated tool such as a theorem prover.

Following the invariant, above the specification of `pop` in Figure 5, are two `initially` clauses.

Following the `initially` clauses are the specifications of the `pop`, `push`, and `top` methods. These are interesting for several new features that they present. Each of these has its behavioral specification written using two specification cases, separated by the keyword `also`. The semantics is that, when the precondition of a case is satisfied, the rest of that case's specification must be obeyed. In these three examples, case analysis is only used to separate the specification of the normal case (the first of the two in each method's specification) from the case where an exception in thrown. In the normal case, `returns` is true, whereas when an exception is thrown `throws(BoundedStackException)` is true.

A specification with several cases is shorthand for one in which the cases are combined [4, 15, 44, 41]. In Figure 6 we show the expanded specification of `pop` from Figure 5. As can be seen from this example, the expanded specification has a postcondition that is a conjunction of implications, one for each case. The implication for a case in the expanded postcondition says that when the precondition for that case holds, the case's postcondition must also hold. The `modifiable` clause for the expanded specification is the union of all the modifiable clauses for the cases; because of this the variables that are named in the

[5] Of course, one could specify `BoundedStack` without separating out the interface for `BoundedThing`, and in that case, this abstraction would be unnecessary. We have made this separation partly to demonstrate more advanced features of JML, and partly to fit the figures on single pages.

```
package edu.iastate.cs.jml.samples.stacks;
//@ model import edu.cs.iastate.jml.models.*;
public interface BoundedStackInterface extends BoundedThing {
  //@ public model JMLObjectSequence theStack;

  //@ public depends size on theStack;
  //@ public represents size by size == theStack.length();
  //@ public invariant redundantly theStack.length() <= MAX_SIZE;

  //@ public initially theStack.isEmpty();
  //@ public initially redundantly theStack.equals(new JMLObjectSequence());

  public void pop( ) throws BoundedStackException;
    //@ behavior {
    //@   requires !theStack.isEmpty();
    //@   modifiable size, theStack;
    //@   ensures returns && theStack.equals(old(theStack.trailer()));
    //@   ensures redundantly theStack.length() == old(theStack.length()) - 1;
    //@  also
    //@   requires theStack.isEmpty();
    //@   ensures throws(BoundedStackException);
    //@ }
  public void push(Object x ) throws BoundedStackException;
    //@ behavior {
    //@   requires theStack.length() < MAX_SIZE;
    //@   modifiable size, theStack;
    //@   ensures returns && theStack.equals(old(theStack.addFirst(x)));
    //@   ensures redundantly theStack.length() == old(theStack.length()) + 1;
    //@  also
    //@   requires theStack.length() == MAX_SIZE;
    //@   ensures throws(BoundedStackException);
    //@ }
  public Object top( ) throws BoundedStackException;
    //@ behavior {
    //@   requires !theStack.isEmpty();
    //@   ensures returns && result == theStack.first();
    //@  also
    //@   requires theStack.isEmpty();
    //@   ensures throws(BoundedStackException);
    //@ }
}
```

Figure 5: A specification of bounded stacks (file `BoundedStackInterface.java`).

```
public void pop( ) throws BoundedStackException;
  //@ behavior {
  //@   requires !theStack.isEmpty() || theStack.isEmpty();
  //@   modifiable size, theStack;
  //@   ensures (!theStack.isEmpty() =>
  //@              returns && theStack.equals(old(theStack.trailer())))
  //@         && (theStack.isEmpty() =>
  //@                    throws(BoundedStackException)
  //@                    && unmodified(size, theStack));
  //@   ensures redundantly !theStack.isEmpty() =>
  //@             theStack.length() == old(theStack.length()) - 1;
  //@ }
```

Figure 6: An expansion of `pop`'s specification. The precondition reduces to `true`, but the precondition shown is the general form for the expansion.

combined modifiable clause but not allowed to be modified in a particular case have to be asserted to be unmodified in that case. In this expansion, the model fields `size` and `theStack` are asserted to be unmodified in the second case's translation.

The `depends` clause is important in "loosening up" the `modifiable` clause, for example to permit the fields of an object that implement the abstract model to be changed [21, 20]. This "loosening up" also applies to model fields that have dependencies declared. For example, since `size` depends on `theStack`, i.e., `size` is in some sense represented by `theStack`, if `size` is mentioned in a `modifiable` clause, then `theStack` is implicitly allowed to be modified. Thus it is only for rhetorical purposes that we mention both `size` and `theStack` in the modifiable clauses of `pop` and `push`. Note, however, that just mentioning `theStack` would not permit `size` to be modified, because `theStack` does not depend on `size`.

Finally, there is more redundancy in the specifications of `pop` and `push`, which each have a redundant `ensures` clause in their normal case. For a redundant `ensures` clause, what one checks is that the conjunction of the precondition, the meaning of the `modifiable` clause, and the (non-redundant) postcondition together imply the redundant postcondition. It is interesting to note that the specifications for stacks written in Eiffel [29, page 339] expresses not much more than what we specify in the redundant postconditions of `pop` and `push`. These convey strictly less information than the non-redundant postconditions, since they say little about the elements of the stack.[6]

2.3 Making New Pure Types

JML comes with a suite of pure types, implemented as Java classes. At the time of this writing these are `JMLObjectSet`, `JMLObjectSequence`, `JMLObjectMap` and `JMLValueSet`, `JMLValueSequence`, `JMLValueMap`, `JMLInteger`, and a few helper classes (such as exceptions

[6] Meyer's specification actually says what the top element of the stack is after a `push`, but says nothing about the rest of the elements. Meyer's second specification and implementation of stacks [29, page 349] is no better in this respect, although, of course, the implementation does keep track of the elements properly.

and enumerators). These are found in the package `edu.iastate.cs.jml.models`, and can be used for defining abstract models. Users can also create their own pure types if desired. Since these types are to be treated as purely immutable values in specifications, they must pass certain conservative checks that make sure there is no possibility of observable side-effects from using such objects.

The extension mechanism uses the modifier `pure`, as in Figure 7. A pure interface must have a specification such that:

- all the methods in each interface it extends are pure (these may be either in pure interfaces or the methods may be explicitly specified as pure),
- all the methods it specifies must be pure in the sense described below.

We say a method or constructor is *pure* if it is either specified with the modifier `pure` or appears in the specification of a `pure` interface or class.

A method (not a constructor) that is pure must have a specification such that:

- it modifies nothing,
- it must terminate when called in a state that satisfies its precondition, and
- it cannot throw an exception that is a subtype of `Error`.

A constructor that is pure must have a specification such that:

- it modifies only the non-static fields of the class in which it appears (including those inherited from its superclasses),
- it must terminate when called in a state that satisfies its precondition, and
- it cannot throw an exception that is a subtype of `Error`.

Implementations of pure methods and constructors will be checked to see that they meet these conditions. In particular, a pure method or constructor implementation is prohibited from calling methods or constructors that are not pure. It must also be provably terminating.

A pure method or constructor can be declared in any class. JML will specify many of the intuitively pure methods and constructors in the standard Java libraries as pure.

A *pure* class must have a specification such that:

- it only extends other pure classes,
- all the methods in each interface it implements extends are pure,
- all its methods and constructors must be specified to be pure in the sense described above, and
- all its data fields must be of some primitive value type or a pure type.

Recursion is permitted, both in pure methods and in data members of pure classes. However, remember that a pure method must be proved to terminate when its preconditions is met.

Model classes should also be pure, since there is no way to use non-pure operations in an assertion. However, the modifiers `model` and `pure` are orthogonal, and thus usually one

will to list both of them when declaring a model class. In particular, one may specify a pure class that is not a model class; such a class would have to be implemented.

As an example, we specify a pure interface, `Money`, that would be suitable for use in abstract models. Our specification is rather artificially broken up into pieces to allow each piece to have a specification that fits on a page. This organization is not necessarily something we would recommend, but it does give us a chance to illustrate more features of JML.

Consider first the interface `Money` specified in Figure 7. The abstract model here is a single field of the primitive Java type `long`, which holds a number of pennies.

This interface has a history constraint, which says that the number of pennies in an object cannot change.[7]

The interesting aspect of the operations is another kind of redundancy, given by the `example` clauses [14, 17]. Any number of examples can be given for a specification case. Here there are three examples in the specification of `dollars` and two in the specification of `cents`. An example's predicate should, when conjoined with any precondition and the modifiable clause should imply the post-condition given. (Note that this is the opposite direction of implication from a redundant ensures clause.) Typically, examples are concrete, and serve to point out various rhetorical points about the use of the specification to the reader. (Exercise: check all the examples given!)

The interface `Money` is specified to extend the interface `JMLType`. This interface is given in Figure 8. It says that objects should have `equals` and `clone` methods.

The specification of `JMLType` is noteworthy in its use of informal predicates [14]. In this instance, the informal predicates are used as an escape from formality. The use of informal predicates avoids the delicate issues of saying what observable aliasing means[8], and what equality of values means.

As specified in Figure 7, the type `Money` lacks some useful operations. The extensions in Figures 9 and 10 provide specifications of comparison operations and arithmetic, respectively.

The specification in Figure 9 is interesting because each of the specified preconditions protects the postcondition from undefinedness in the postcondition [18]. For example, if the argument `m2` in the `greaterThan` method were `null`, then the expression `m2.pennies` would not be defined.

The specification of `MoneyOps` in Figure 10 is interesting for the use of a model method, `inRange`. This method cannot be invoked by Java programs; that is, it would not appear in the Java implementation. When used in a predicate, `inRange(l)` is equivalent to using some correct implementation of its specification. The specification of `inRange` also makes use of a local model variable declaration. Such declarations allow one to abbreviate long expressions, or, to make rhetorical points by naming constants, as is done with `epsilon`.

Note also that JML uses the Java semantics for mixed-type expressions; for example in the specification of `plus` in Figure 10, `m2.pennies` is coerced to a double-precision floating point number, as it would be in Java.

[7] There is no `initially` clause in this interface, so data type induction cannot assume any particular starting value. But this is desirable, since if a particular starting value was specified, then by the history constraint, all objects would have that value.

[8] *Observable aliasing* is a sharing relation between objects that can be detected by a program. Such a program, might, for example modify one object and read a changed value from the shared object. Formalizing this is a bit beyond what we wish to describe at this point.

```
package edu.iastate.cs.jml.docs.prelimdesign;

import edu.iastate.cs.jml.models.JMLType;

public /*@ pure @*/ interface Money extends JMLType
{
  //@ public model long pennies;

  //@ public constraint pennies == old(pennies);

  public long dollars();
  //@ behavior {
  //@   ensures result == pennies / 100;
  //@   example pennies == 703 && result == 7;
  //@   example pennies == 799 && result == 7;
  //@   example pennies == -503 && result == -5;
  //@ }

  public long cents();
  //@ behavior {
  //@   ensures result == pennies % 100;
  //@   example pennies == 703 && result == 3;
  //@   example pennies == -503 && result == -3;
  //@ }

  public boolean equals(Object o2);
  //@ behavior {
  //@   ensures result == (o2 instanceof Money
  //@                       && pennies == (Money)o2.pennies);
  //@ }

  public Object clone();
  //@ behavior {
  //@   ensures result instanceof Money
  //@           &&(Money)result.pennies == pennies;
  //@ }
}
```

Figure 7: A specification of the pure interface `Money` (file `Money.java`).

```
// @(#)$Id: JMLType.java,v 1.17 1998/11/04 04:47:58 leavens Exp $

package edu.iastate.cs.jml.models;

public interface JMLType extends Cloneable {

  public /*@ pure @*/ Object clone();
  /*@ behavior {
    @   ensures result instanceof JMLType
    @           && ((JMLType)result).equals(this)
    @           && informally (* result and this are not observably aliased *);
    @ }
    @*/

  public /*@ pure @*/ boolean equals(Object op2);
  /*@ behavior {
    @   ensures result =>
    @              (informally (* op2 is an instance of this.getClass() *)
    @               && informally (* op2 is not distinguishable from this *));
    @ }
    @*/
}
```

Figure 8: A specification of the interface `JMLType` (file `JMLType.java`).

```
package edu.iastate.cs.jml.docs.prelimdesign;

public /*@ pure @*/ interface MoneyComparable extends Money
{
  public boolean greaterThan(Money m2);
  //@ behavior {
  //@   requires m2 != null;
  //@   ensures result == (pennies > m2.pennies);
  //@ }

  public boolean greaterThanOrEqualTo(Money m2);
  //@ behavior {
  //@   requires m2 != null;
  //@   ensures result == (pennies >= m2.pennies);
  //@ }

  public boolean lessThan(Money m2);
  //@ behavior {
  //@   requires m2 != null;
  //@   ensures result == (pennies < m2.pennies);
  //@ }

  public boolean lessThanOrEqualTo(Money m2);
  //@ behavior {
  //@   requires m2 != null;
  //@   ensures result == (pennies <= m2.pennies);
  //@ }
}
```

Figure 9: A pure interface specification (file `MoneyComparable.java`).

```
package edu.iastate.cs.jml.docs.prelimdesign;

public /*@ pure @*/ interface MoneyOps extends MoneyComparable
{
  //@ model public boolean inRange(double d);
  //@ behavior {
  //@   model double epsilon = 1.0;
  //@   ensures result == (Long.MIN_VALUE + epsilon < d
  //@                       && d < Long.MAX_VALUE - epsilon);
  //@ }

  public Money plus(Money m2);
  //@ behavior {
  //@   requires m2 != null && inRange((double) pennies + m2.pennies);
  //@   ensures result != null
  //@           && result.pennies == this.pennies + m2.pennies;
  //@   example this.pennies == 300 && m2.pennies == 400
  //@           && result.pennies == 700;
  //@ }

  public Money minus(Money m2);
  //@ behavior {
  //@   requires m2 != null && inRange((double) pennies - m2.pennies);
  //@   ensures result != null
  //@           && result.pennies == this.pennies - m2.pennies;
  //@   example this.pennies == 400 && m2.pennies == 300
  //@           && result.pennies == 100;
  //@ }

  public Money scaleBy(double factor);
  //@ behavior {
  //@   requires inRange(factor * pennies);
  //@   ensures result != null
  //@           && result.pennies == (long)(factor * pennies);
  //@   example pennies == 400 && factor == 1.01
  //@           && result.pennies == 404;
  //@ }
}
```

Figure 10: A specification of the pure interface `MoneyOps` (file `MoneyOps.java`).

2.4 Implementation of Class and Interface Specifications

The key to proofs that an implementation of a class or interface specification is correct lies in the use of `depends` and `represents` clauses [11, 21]. Consider, for example, the abstract class `MoneyAC` given in Figure 11. This class is abstract and has no constructors. The class declares a concrete field `numCents`, which is related to the model field `pennies` by the `represents` clause. This allows relatively trivial proofs of the correctness of the `dollars` and `cents` methods, and is key to the proofs of the other methods.

The straightforward implementation of the subclass `MoneyComparableAC` is given in Figure 12. Note that the model and concrete fields are both inherited by this class.

An interesting feature of the class `MoneyComparableAC` is the protected static method `totalCents`. For this method, we give its code with an embedded assertion. Such assertions are an alternative to the usual behavioral specification format in JML.

Note that the model method, `inRange` is not implemented, and does not need to be implemented to make this class correctly implement the interface `MoneyComparable`.

Finally, a concrete class implementation is the class `USMoney`, presented in Figure 13. This class implements the interface `MoneyOps`. Note that specifications as well as code are given for the constructors.

The first constructor's specification illustrates that redundancy can also be used in a `modifiable` clause. A redundant `modifiable` clause follows if the meaning of the set of locations named is a subset of the ones given in the non-redundant clause for the same specification case. In this example the redundant modifiable clause follows from the given modifiable clause and the meaning of the `depends` clause inherited from the superclass `MoneyAC`.

The second constructor in Figure 13 is noteworthy in that there is a redundant ensures clauses that uses an informal predicate [14]. In this instance, the informal predicate is used as a comment (which could also be used). Recall that informal predicates allow an escape from formality when one does not wish to give part of a specification in formal detail.

2.5 Use of Pure Classes

Since `USMoney` is a pure class, it can be used to make models of other classes. An example is the abstract class `Account` given in Figure 14. The first model field in this class has type `USMoney`.

The specification of `Account` makes good use of examples. It will be used below to describe the requirements for behavioral subtyping.

2.6 Composition for Container Classes

The following example specification of a class `Digraph` (directed graph) gives a more interesting example of the JML way that more complex models are composed from other classes. In this example we use model classes, and the pure containers provided in the package `edu.iastate.cs.jml.models`.

Figure 15 contains an abstract class `NodeType`. `NodeType` is an abstract class (rather than a model class) because it will require an implementation and does appear in the interface of model class `Digraph`. However, we also denote the abstract class as `pure`, since we will also use `NodeType` in the specification of other classes. (And we do so appropriately, since all the methods for class `NodeType` are side-effect-free.) In the abstract class specification for `NodeType` we simply provide a model field `iD`, which would represent a unique

```
package edu.iastate.cs.jml.docs.prelimdesign;

public /*@ pure @*/ abstract class MoneyAC implements Money {

  protected long numCents;
  //@ protected depends pennies on numCents;
  //@ protected represents pennies by pennies == numCents;

  //@ protected constraint redundantly numCents == old(numCents);

  public long dollars()
  {
    return numCents / 100;
  }

  public long cents()
  {
    return numCents % 100;
  }

  public boolean equals(Object o2)
  {
    try {
      Money m2 = (Money)o2;
      return numCents == (100 * m2.dollars() + m2.cents());
    } catch (ClassCastException e) {
      return false;
    }
  }

  public Object clone()
  {
    return this;
  }
}
```

Figure 11: A pure abstract class `MoneyAC` that implements the interface `Money` (file `MoneyAC.java`).

```
package edu.iastate.cs.jml.docs.prelimdesign;

public /*@ pure @*/ abstract class MoneyComparableAC
    extends MoneyAC implements MoneyComparable
{
  protected static long totalCents(Money m2)
  {
    return 100 * m2.dollars() + m2.cents();
    //@ ensures result == m2.pennies;
  }

  public boolean greaterThan(Money m2)
  {
    return numCents > totalCents(m2);
  }

  public boolean greaterThanOrEqualTo(Money m2)
  {
    return numCents >= totalCents(m2);
  }

  public boolean lessThan(Money m2)
  {
    return numCents < totalCents(m2);
  }

  public boolean lessThanOrEqualTo(Money m2)
  {
    return numCents <= totalCents(m2);
  }
}
```

Figure 12: A pure abstract class `MoneyComparableAC` that implements the interface `MoneyComparable` and extends the class `MoneyAC` (file `MoneyComparableAC.java`).

```
package edu.iastate.cs.jml.docs.prelimdesign;

public /*@ pure @*/ class USMoney
                extends MoneyComparableAC implements MoneyOps
{
  public USMoney(long cs)
       //@ behavior {
       //@   modifiable pennies;
       //@   modifiable redundantly numCents;
       //@   ensures pennies == cs;
       //@   ensures redundantly numCents == cs;
       //@ }
  {
    numCents = cs;
  }

  public USMoney(double amt)
       //@ behavior {
       //@   modifiable pennies;
       //@   ensures pennies == (long)(100.0 * amt);
       //@   ensures redundantly informally (* pennies holds amt dollars *);
       //@ }
  {
    numCents = (long)(100.0 * amt);
  }

  public Money plus(Money m2)
  {
    //@ assert m2 != null;
    return new USMoney(numCents + totalCents(m2));
  }

  public Money minus(Money m2)
  {
    //@ assert m2 != null;
    return new USMoney(numCents - totalCents(m2));
  }

  public Money scaleBy(double factor)
  {
    return new USMoney(numCents * factor / 100.0);
  }
}
```

Figure 13: A pure concrete class `USMoney` (file `USMoney.java`).

```
package edu.iastate.cs.jml.docs.prelimdesign;
public class Account {
  public model USMoney credit;
  public model String owner;
  public invariant credit.greaterThanOrEqualTo(new USMoney(0));
  public constraint owner.equals(old(owner));
  public Account(MoneyOps amt, String own);
  behavior {
     requires (new USMoney(1)).lessThanOrEqualTo(amt);
     modifiable credit, owner;
     ensures credit.equals(amt) && owner.equals(own);
  }
  public MoneyOps balance();
  behavior {
     ensures result.equals(credit);
  }
  public void payInterest(double rate);
  behavior {
     requires 0.0 <= rate && rate <= 1.0;
     modifiable credit;
     ensures credit.equals(old(credit).scaleBy(1.0 + rate));
     example rate == 0.05 && old(credit).equals(new USMoney(4000))
          && credit.equals(new USMoney(4200));
   }
  public void deposit(MoneyOps amt);
  behavior {
     requires amt.greaterThanOrEqualTo(new USMoney(0));
     modifiable credit;
     ensures credit.equals(old(credit).plus(amt));
     example old(credit).equals(new USMoney(40000))
          && amt.equals(new USMoney(1))
          && credit.equals(new USMoney(40001));
   }
  public void withdraw(MoneyOps amt);
  behavior {
     requires (new USMoney(0)).lessThanOrEqualTo(amt)
               && amt.lessThanOrEqualTo(credit);
     modifiable credit;
     ensures credit.equals(old(credit).minus(amt));
     example old(credit).equals(new USMoney(40001))
          && amt.equals(new USMoney(40000))
          && credit.equals(new USMoney(1));
   }
}
```

Figure 14: Specification of a pure concrete class `Account` (file `Account.jml`).

```
package edu.iastate.cs.jml.samples.Digraph;

import edu.iastate.cs.jml.models.*;

public /*@ pure @*/ abstract class NodeType implements JMLType {

  //@ public model int iD;

  public abstract boolean equals(Object o);
  //@ behavior {
  //@   requires o instanceof NodeType;
  //@   ensures result == (iD == (NodeType)o.iD);
  //@ also
  //@   requires !(o instanceof NodeType);
  //@   ensures result == false;
  //@ }

  public abstract Object clone();
  //@ behavior {
  //@   ensures result instanceof NodeType
  //@     && ((NodeType)result).equals(this) && fresh(result);
  ///@  ensures redundantly result != this;
  //@ }

}  // end of class NodeType declaration
```

Figure 15: First part of an abstract class specification `NodeType` (file `NodeType.java`).

identifier for nodes. We specify that the `equals` method for class `NodeType` simply tests whether two references to objects of type `NodeType` are references to objects with the same iD's. We also require that `NodeType` have a public `clone` method that behaves as we expect such methods to behave.

Figure 16 contains the specification for a pure model class `ArcType`. We will use `ArcType` in the model for `Digraph`, but `ArcType` does not require an implementation since it does not appear in the interface to `Digraph`. We declare `ArcType` to be a pure class so that its methods can be used in assertions. The two model fields for `ArcType`, `from` and `to`, are both of type `NodeType`. We specify the `equals` method so that two references to objects of type `ArcType` are equal if and only if they have equal values in the `from` and `to` model fields. That is, `equals` is specified using `NodeType.equals`. We specify that `ArcType` support a public `clone` method as it is required for the container we will use for the type of one of the model fields in `Digraph`. We will also make use of the constructor in the specification of `Digraph`.

Finally, the specification of the class `Digraph` in Figures 17, 18, and 19 demonstrates how to use container classes to compose models in JML. Both the model fields `nodes` and `arcs` are of type `JMLValueSet`. However, in the first invariant clause we restrict `nodes`

```
package edu.iastate.cs.jml.samples.Digraph;

import edu.cs.iastate.jml.models.JMLType;

public pure model abstract class ArcType implements JMLType {

  public model NodeType from;
  public model NodeType to;
  public invariant from != null && to != null;

  public ArcType(NodeType inFrom, NodeType inTo);
    behavior {
      requires inFrom != null && inTo != null;
      modifiable from, to;
      ensures fresh(from) && fresh(to)
              && from.equals(inFrom) && to.equals(inTo);
      ensures redundantly from != inFrom && to != inTo;
    }

  public model boolean equals(Object o);
    behavior {
      requires o instanceof ArcType;
      ensures result == (    ((ArcType)o).from.iD == from.iD
                          && ((ArcType)o).to.iD == to.iD );
    also
      requires !(o instanceof ArcType);
      ensures result == false;
    }

  public Object clone();
    behavior {
      ensures result instanceof ArcType && fresh(result)
              && ((ArcType)result).equals(this);
      ensures redundantly result != this;
    }
}
```

Figure 16: First part of a model class specification `ArcType` (file `ArcType.jml`).

```
package edu.iastate.cs.jml.samples.Digraph;
model import edu.cs.iastate.jml.models.*;
public class Digraph {

 public model JMLValueSet nodes;
 public model JMLValueSet arcs;

 public invariant forall (JMLType n)
                     [ nodes.isIn(n) => n instanceof NodeType];
 public invariant forall (JMLType a)
                     [ arcs.isIn(a) => a instanceof ArcType];
 public invariant forall (ArcType a)
                     [ arcs.isIn(a) =>
                          nodes.isIn(a.from) && nodes.isIn(a.to) ];

 public Digraph();
   behavior {
     modifiable nodes, arcs;
     ensures nodes.equals(new JMLValueSet())
             && arcs.equals(new JMLValueSet());
   }
```

Figure 17: First part of a class specification `Digraph` (file `Digraph.jml`).

so that every object in `nodes` is, in fact, of type `NodeType`. Similarly, the next invariant clause we restrict `arcs` to be a set of `ArcType` objects. In both cases, since we are using `JMLValueSet`, membership is determined by the use of the `equals` method for the type of the elements (rather than reference equality).

Thus, in JML, one uses containers like `JMLValueSet`, combined with appropriate invariants to specify models that are compositions of other classes. These classes are typically pure, which means that all their methods are side-effect free (see below), making them suitable for use in assertions.

An interesting use of pure model methods appears in Figure 19. The pure model method `ReachSet` constructively defines the set of all nodes that are reachable from the nodes in the argument `nodeSet`. Note the recursive `ensures` clause for `ReachSet`, which builds up the entire set of reachable nodes by, for each recursive reference, adding the nodes that can be reached directly (via a single arc) from the nodes in `nodeSet`. Such recursive definitions must, to be well-defined, be provably terminating, which is the purpose of the `measured by` clause in the specification of `ReachSet`. This clause defines an integer-valued measure that must always be at least zero; furthermore, the measure for a call and recursive uses in the specification must strictly decrease [32].

```
public void addNode(NodeType n);
  behavior {
    requires n != null;
    modifiable nodes;
    ensures nodes.equals(old(nodes.insert(n)));
  }

public void removeNode(NodeType n);
  behavior {
    requires unconnected(n);
    modifiable nodes;
    ensures nodes.equals(old(odes.remove(n)));
  }

public void addArc(NodeType inFrom, NodeType inTo);
  behavior {
    requires inFrom != null && inTo != null
             && nodes.isIn(inFrom) && nodes.isIn(inTo);
    modifiable arcs;
    ensures arcs.equals(old(arcs.insert(new ArcType(inFrom, inTo))));
  }

public pure boolean isNode(NodeType n);
  behavior {
    ensures result == (nodes.isIn(n));
  }

public pure boolean isArc(NodeType inFrom, NodeType inTo);
  behavior {
    ensures result == (arcs.isIn(new ArcType(inFrom, inTo)));
  }

public pure boolean isAPath(NodeType start, NodeType end);
  behavior {
    requires nodes.isIn(start) && nodes.isIn(end);
    ensures result == ReachSet(new JMLValueSet().insert(start)).isIn(end);
  }
```

Figure 18: Second part of a class specification `Digraph` (file `Digraph.jml`, continued).

```
 public pure model boolean unconnected(NodeType n);
   behavior {
     ensures result == !exists (ArcType a) [ arcs.isIn(a)
                         && (a.from.equals(n) || a.to.equals(n)) ];
   }

 public pure model JMLValueSet ReachSet(JMLValueSet nodeSet);
   behavior {
     requires nodeSet != null
       && forall (Object o) [nodeSet.isIn(o) =>
            o instanceof NodeType && nodes.isIn(o)];
     measured by nodes.size() - nodeSet.size();
     ensures
       (nodeSet.equals(OneMoreStep(nodeSet)) => result.equals(nodeSet))
       && (!nodeSet.equals(OneMoreStep(nodeSet)) =>
             result.equals(ReachSet(OneMoreStep(nodeSet))) );
   }

 public pure model JMLValueSet OneMoreStep(JMLValueSet nodeSet);
   behavior {
     requires nodeSet != null
       && forall (Object o) [nodeSet.isIn(o) =>
            o instanceof NodeType && nodes.isIn(o)];
     ensures result.equals(nodeSet.union(
       new JMLValueSet { NodeType n |
          exists (ArcType a) [arcs.isIn(a) &&
            (    nodeSet.isIn(a.from) && n.equals(a.to)
              || nodeSet.isIn(a.to) && n.equals(a.from) )]}));
   }
}  // end of class Digraph
```

Figure 19: Third part of a class specification `Digraph` (file `Digraph.jml`, continued).

2.7 Subtyping

Following Dhara and Leavens [3, 15], a subtype inherits the specifications of its supertype's public and protected members (fields and methods), as well as invariants and history constraints. This ensures that a subclass specifies a behavioral subtype of its supertypes; This inheritance can be thought of textually, by copying the specifications of the methods of a class's ancestors and all interfaces that a class implements into the class's specification as specification cases; these cases must be satisfied by the method, in addition to any explicitly specified cases,

For example, consider `PlusAccount` as a subclass of `Account`. It inherits the fields from `Account`, and the initially clauses, invariants, and history constraints from `Account`. Because it inherits the fields of its superclass, inherited method specifications are still meaningful when copied to the subclass. The trick is to always add new model fields to the subclass and relate them to the existing ones.

3 Extensions to Java Expressions for Predicates

The expressions that can be used as predicates in JML are an extension to the side-effect free Java expressions. Since predicates are required to be side-effect free, the following Java operators are *not* allowed within predicates:

- assignment (`=`), and the various assignment operators (such as `+=`, `-=`, etc.)
- all forms of increment and decrement operators (`++` and `--`), and
- calls to methods that are not pure.

We allow the allocation of storage (e.g., using operator `new` and pure constructors) in predicates, because such storage can never be referred to after the evaluation of the predicate, and because such pure constructors have no side-effects other than initializing the new objects so created

JML adds the following new syntax to the Java expression syntax, for use in predicates:

- `=>` for logical implication; for example, `raining => getsWet` is true if either `raining` is false or `getsWet` is true.
- `forall` and `exists`, which are quantifiers; for example,

  ```
  forall (int i,j) [0 <= i && i < j && j < 10 => a[i] < a[j] ]
  ```

  says that `a` is sorted at indexes between 0 and 9.
- `returns`, which is true if a method returns normally, when an exception is thrown this is false.
- `result`, which, when `returns` is true, is the object that is the result of the method.
- `throws`, which can be used to assert that a particular exception is thrown; for example `throws(ArithmeticException)` is true when the exception `ArithmeticException` is thrown.

```
package edu.iastate.cs.jml.docs.prelimdesign;
public class PlusAccount extends Account {
  public model USMoney savings, checking;

  public depends credit on savings, checking;
  public represents credit by credit.equals(savings.plus(checking));

  public invariant redundantly savings.plus(checking) >= new USMoney(0);

  public PlusAccount(MoneyOps sav, MoneyOps chk, String own);
  behavior {
     requires (new USMoney(1)).lessThanOrEqualTo(sav)
           && (new USMoney(1)).lessThanOrEqualTo(chk);
     modifiable credit, owner;
     modifiable redundantly savings, checking;
     ensures savings.equals(sav) && checking.equals(chk)
             && owner.equals(own);
     ensures redundantly credit.equals(amt);
  }

  public void payInterest(double rate);
  behavior {
     requires 0.0 <= rate && rate <= 1.0;
     modifiable credit, savings, checking;
     ensures checking.equals(old(checking).scaleBy(1.0 + rate));
     example rate == 0.05 && old(checking).equals(new USMoney(2000))
          && checking.equals(new USMoney(2100));
  }

  public void deposit(MoneyOps amt);
  behavior {
     requires amt.greaterThanOrEqualTo(new USMoney(0));
     modifiable credit, savings;
     ensures savings.equals(old(savings).plus(amt));
     ensures redundantly unchanged(checking);
     example old(savings).equals(new USMoney(20000))
          && amt.equals(new USMoney(1))
          && savings.equals(new USMoney(20001));
  }
```

Figure 20: The class `PlusAccount`, (file `PlusAccount.jml`, first part).

```
public void withdraw(MoneyOps amt);
behavior {
   requires (new USMoney(0)).lessThanOrEqualTo(amt)
            && amt.lessThanOrEqualTo(savings);
   modifiable credit, savings;
   ensures savings.equals(old(savings).minus(amt));
   ensures redundantly unmodified(checking);
   example old(savings).equals(new USMoney(40001))
        && amt.equals(new USMoney(40000))
        && savings.equals(new USMoney(1));
  also
   requires (new USMoney(0)).lessThanOrEqualTo(amt)
            && amt.lessThanOrEqualTo(credit)
            && amt.greaterThan(savings);
   modifiable credit, savings, checking;
   ensures savings.equals(new USMoney(0))
           && checking.equals(old(checking).minus(amt.minus(savings)));
   example old(savings).equals(new USMoney(30001))
        && old(checking).equals(new USMoney(10000))
        && amt.equals(new USMoney(40000))
        && savings.equals(new USMoney(0))
        && checking.equals(new USMoney(1));
}

public void depositToChecking(MoneyOps amt);
behavior {
   requires amt.greaterThanOrEqualTo(new USMoney(0));
   modifiable credit, checking;
   ensures checking.equals(old(checking).plus(amt)) && unchanged(savings);
   example old(checking).equals(new USMoney(20000))
        && amt.equals(new USMoney(1))
        && checking.equals(new USMoney(20001));
}
```

Figure 21: The class `PlusAccount`, continued (file `PlusAccount.jml`, second part).

```
  public void payCheck(MoneyOps amt);
  behavior {
     requires (new USMoney(0)).lessThanOrEqualTo(amt)
              && amt.lessThanOrEqualTo(checking);
     modifiable credit, checking;
     ensures checking.equals(old(checking).minus(amt));
     example old(checking).equals(new USMoney(40001))
          && amt.equals(new USMoney(40000))
          && checking.equals(new USMoney(1));
    also
     requires (new USMoney(0)).lessThanOrEqualTo(amt)
              && amt.lessThanOrEqualTo(credit)
              && amt.greaterThan(checking);
     modifiable credit, checking, savings;
     ensures checking.equals(new USMoney(0))
             && savings.equals(old(savings).minus(amt.minus(checking)));
     example old(savings).equals(new USMoney(30001))
          && old(checking).equals(new USMoney(10000))
          && amt.equals(new USMoney(40000))
          && checking.equals(new USMoney(0))
          && savings.equals(new USMoney(1));
 }
}
```

Figure 22: The class `PlusAccount`, continued (file `PlusAccount.jml`, last part).

- `thrown`, which can be used to describe the object that is the "exception result" when a method throws an exception; for example when `throws(ArithmeticException)` is true, then `thrown(ArithmeticException)` is the object that was thrown.
- `fresh`, which asserts that objects were freshly allocated; for example, `fresh(x,y)` asserts that the objects bound to `x` and `y` were not allocated in the pre-state.
- `old`, which can be used to refer to values in the pre-state; e.g., `old(myPoint.x)` is the value of the `x` field of the object `myPoint` in the pre-state.
- `unmodified`, which asserts that the values of objects are the same in the post-state as in the pre-state; for example, `unmodified(xval,yval)` says that `xval` and `yval` have the same value in the pre- and post-states (in the sense of an `equals` method).
- `reach`, which returns a `JMLObjectSet` of all objects reachable from a given object.
- Set comprehensions, which can be used to succinctly define sets; for example, the following is the `JMLObjectSet` of `Integer` objects whose values are between 0 and 10, inclusive.

```
new JMLObjectSet {Integer i | 0 <= i.getInteger()
                            && i.getInteger() <= 10 }
```

As in Java itself, most types are reference types, and hence many expressions yield references (i.e., object identities or addresses), as opposed to primitive values. This means that `==`, except when used to compare pure values of primitive types such as `boolean` or `int`, is reference equality. As in Java, to get value equality, except for primitive values, one has to use the `equals` method in assertions. For example, the predicate `myString == yourString`, is only true if the objects denoted by `myString` and `yourString` are the same object (i.e., if the names are aliases); to compare the values of two strings one must write `myString.equals(yourString)`.

The reference semantics makes interpreting predicates that involve the use of `old` interesting. We want to have the semantics suited for two purposes:

- execution of assertions for purposes of debugging and testing, as in Eiffel, and
- generation of mathematical assertions for static analysis and possible theorem proving (e.g., to verify program correctness).

The key to the semantics of `old` is to treat it as an abbreviation for a local definition. That is, E in `old(`E`)` can be evaluated in the pre-state, and its value bound to a locally defined name, and then the name can be used in the postcondition.

Since we are using Java expressions for predicates, there are some additional problems in mathematical modeling. We are excluding the possibility of side-effects by limiting the syntax of predicates, and by using type checking [6, 24, 25, 30, 36, 45] to make sure that only pure methods and constructors may be called in predicates.

Exceptions in expressions are particularly important, since they may arise in type casts. Logically, we will deal with exceptions by having the evaluation of predicates substitute an arbitrary expressible value of the normal result type when an exception is thrown during evaluation. (When the expression's result type is a reference type, an implementation would have to return `null` if an exception is thrown while executing such a predicate.)

This corresponds to a mathematical model in which partial functions are mathematically modeled by underspecified total functions.

We will check that errors (i.e., exceptions that inherit from Error) are not explicitly thrown by pure methods. This means that they can be ignored during mathematical modeling. When executing predicates, errors will also be ignored, but will cause run time errors.

4 Conclusions

One area of future work for JML is concurrency. Our current plan is to use `when` clauses that say when a method may proceed to execute, after it is called [22, 34]. This permits the specification of when the caller is delayed to obtain a lock, for example. While syntax for this exists in the JML parser, our exploration of this topic is still in an early stage.

Acknowledgements

Thanks to Rustan Leino and Peter Mueller for many discussions about the semantics of such specifications. For comments on an earlier draft, thanks to Peter, Anand Ganapathy, Sevtap Oltes, Gary Daugherty, Karl Hoech, Jim Potts, Tammy Scherbring, Joachim van den Berg.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[2] Alex Borgida, John Mylopoulos, and Rayomnd Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[3] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996.

[4] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. Technical Report 95-20c, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 1997. Also in Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996, pp. 258–267. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[5] Kate Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.

[6] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, pages 28–38. ACM, August 1986.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[8] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, N.Y., 1993.

[9] I. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, Inc., second edition, 1993.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[11] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[12] Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[13] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies*. The Object-Oriented Series. Prentice Hall, New York, N.Y., 1994.

[14] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

[15] Gary T. Leavens. Larch/C++ Reference Manual. Version 5.14. Available in ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz or on the World Wide Web at the URL http://www.cs.iastate.edu/~leavens/larchc++.html, October 1997.

[16] Gary T. Leavens. Larch frequently asked questions. Version 1.89. Available in http://www.cs.iastate.edu/~leavens/larch-faq.html, January 1998.

[17] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. Technical Report 97-19, Iowa State University, Department of Computer Science, September 1997.

[18] Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 520–534. Springer-Verlag, New York, N.Y., 1997.

[19] Henry. F. Ledgard. A human engineered variant of bnf. *ACM SIGPLAN Notices*, 15(10):57–62, October 1980.

[20] K. Rustan M. Leino. A myth in the modular specification of programs. Technical Report KRML 63, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, November 1995. Obtain from the author, at rustan@pa.dec.com.

[21] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[22] Richard Allen Lerner. Specifying objects of concurrent systems. Ph.D. Thesis CMU-CS-91-131, School of Computer Science, Carnegie Mellon University, May 1991.

[23] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[24] John M. Lucassen. Types and effects: Towards the integration of functional and imperative programming. Technical Report TR-408, Massachusetts Institute of Technology, Laboratory for Computer Science, August 1987.

[25] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif.*, pages 47–57. ACM, January 1988.

[26] David Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA - A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, N.Y., 1987.

[27] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[28] Bertrand Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[29] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., second edition, 1997.

[30] H. R. Nielson, F. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In M. Dam, editor, *Proceedings of the Fifth LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science. Springer-Verlag, 1997.

[31] William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct 1994.

[32] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[33] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[34] Gowri Sivaprasad. Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces. Technical Report 95-27a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 1995.

[35] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, New York, N.Y., second edition, 1992.

[36] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[37] Yang Meng Tan. Interface language for supporting programming styles. *ACM SIGPLAN Notices*, 29(8):74–83, August 1994. Proceedings of the Workshop on Interface Definition Languages.

[38] Yang Meng Tan. *Formal Specification Techniques for Engineering Modular C Programs*, volume 1 of *Kluwer International Series in Software Engineering*. Kluwer Academic Publishers, Boston, 1995.

[39] Tim Wahls, Gary T. Leavens, and Albert L. Baker. Executing formal specifications with constraint programming. Technical Report 97-12a, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, August 1998. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[40] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International Series in Computer Science. Prentice-Hall, New York, N.Y., 1991.

[41] Alan Wills. Refinement in Fresco. In Lano and Houghton [13], chapter 9, pages 184–201.

[42] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[43] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, September 1990.

[44] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[45] Andrew K. Wright. Typing references by effect inference. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer-Verlag, New York, N.Y., 1992.

A Syntax

We use an extended BNF grammar to describe the syntax of Larch/C++. The extensions are as follows [19].

- Nonterminal symbols are written as follows: *nonterminal*.
- Terminal symbols are written as follows: `terminal`. In a few cases it is also necessary to quote terminal symbols, such as when using '`|`' as a terminal symbol instead of a meta-symbol.
- Square brackets ([and]) surround optional text. Note that '`[`' and '`]`' are terminals.
- The notation ... means that the preceding nonterminal or group of optional text can be repeated zero (0) or more times.

compilation-unit ::= [*package-definition*]
[*refine-prefix*]
[*import-definition*] ...
[*type-or-uses*] ...
package-definition ::= `package` *identifier* `;`
refine-prefix ::= `refine` *string-literal* `;`
import-definition ::= `import` *identifier-star* `;`
| `model import` *identifier-star* `;`
identifier ::= *ident* [`.` *ident*] ...
identifier-star ::= *ident* [`.` *ident*] ... [`.` `*`]
type-or-uses ::= [*doc-comment*] *type-definition*
| *uses-clause*

Figure 23: Syntax of compilation units.

For example, the following gives a production for the nonterminal *identifier*, which is a list of *ident*'s separated by periods (`.`).

identifier ::= *ident* [`.` *ident*] ...

To remind the reader that the notation '...' means zero or more repetitions, we use '...' only following optional text.

We use "//" to start a comment (to you, the reader) in the grammar.

A.1 Context-Free Syntax

Figure 23 gives the syntax of compilation units in JML.

Figure 24 gives the syntax of type definitions.

Figure 25 gives the syntax of behavioral specifications for types.

Figure 26 gives the syntax of behavioral specifications for methods.

Figures 27 and 28 give the syntax of predicates and predicate expressions. The precedence of operators in JML is similar to that in Java The precedence levels are given in Table 1.

Figure 29 gives the syntax of the uses clause.

Figure 30 gives the syntax of statements and assertions.

Figure 31 gives the syntax of expressions.

A.2 Microsyntax (Lexical Grammar)

Throughout this section, grammatical productions are to be understood lexically; that is, this grammar concerns individual characters, not tokens. Another way of thinking of this is that no *white-space* may intervene between the characters of a token.

The microsyntax of JML is described by the production *microsyntax* in Figure 33; it describes what a program looks like from the point of view of a lexical analyzer [40].

type-definition ::= *modifiers class-or-interface-def*
| ;
class-or-interface-def ::= *class-definition*
| *interface-definition*
type-spec ::= *type* [*dims*]
type ::= *identifier* | *builtInType*
modifiers ::= [*modifier*] ...
modifier ::= `private` | `public` | `protected`
| `static` | `transient` | `final`
| `abstract` | `native` | `threadsafe`
| `synchronized` | `const` | `volatile`
| `model` | `pure`
class-definition ::= `class` *ident* [`extends` *identifier* [`weakly`]]
[*implements-clause*] *class-block*
class-block ::= `{` [*field*] ... `}`
interface-extends ::= `extends` *identifier-list*
implements-clause ::= `implements` *identifier-list*
identifier-list ::= *identifier* [`weakly`] [`,` *identifier* [`weakly`]] ...
field ::= [*doc-comment*] *modifiers member-decl*
| [`static`] *compound-statement*
| *modifiers initially*
| *modifiers invariant*
| *modifiers history-constraint*
| *modifiers depends-decl*
| *modifiers represents-decl*
| ;
| *uses-clause*
member-decl ::= *variable-decls* `;` | *method-decl*
| *class-definition* | *interface-definition*
variable-decls ::= *type-spec variable-declarators*
variable-declarators ::= *variable-declarator* [`,` *variable-declarator*] ...
variable-declarator ::= *ident* [*dims*] [`=` *initializer*]
initializer ::= *expression* | *array-initializer*
array-initializer ::= `{` [*initializer-list*] `}`
initializer-list ::= *initializer* [`,` *initializer*] ... [`,`]
method-decl ::= [*type-spec*] *method-head method-body*
method-head ::= *ident* `(` [*param-declaration-list*] `)` [*dims*] *throws-clause*
method-body ::= [*behavior*] *compound-statement*
| `;` [*behavior*]
throws-clause ::= `throws` *identifier* [`,` *identifier*] ...
param-declaration-list ::= *param-declaration* [`,` *param-declaration*] ...
param-declaration ::= [`final`] *type-spec ident* [*dims*]

Figure 24: Syntax of type definitions

initially ::= `initially` [`redundantly`] *predicate* `;`
invariant ::= `invariant` [`redundantly`] *predicate* `;`
history-constraint ::= `constraint` [`redundantly`] *predicate*
[`for` *constrained-set*] `;`
constrained-set ::= *method-name* [`,` *method-name*] ...
| `everything`
method-name ::= *identifier* [`(` [*param-disambig-list*] `)`]
param-disambig-list ::= *param-disambig* [`,` *param-disambig*] ...
param-disambig ::= *type-spec* [*ident* [*dims*]]
depends-decl ::= `depends` [`redundantly`] *store-ref* `on` *store-ref-list* `;`
represents-decl ::= `represents` [`redundantly`] *store-ref* `by` *predicate* `;`
store-ref-list ::= *store-ref* [`,` *store-ref*] ...
| `nothing`
| `everything`
store-ref ::= *pred-expression*
| `reach` `(` *pred-expression* `)`

Figure 25: Syntax of behavioral specifications for types.

highest	`new () old fresh throws thrown unmodified forall exists`
	`[]` `.` and method calls
	`+` (unary) `-` (unary) `! (typecast) instanceof`
	`* / %`
	`+` (binary) `-` (binary)
	`<< >> >>>`
	`< <= > >= informally`
	`== !=`
	`&`
	`^`
	`|`
	`&&`
	`||`
	`=>`
	`<=>`
lowest	`?:`

Table 1: Table of operator precedence in JML.

```
behavior ::= behavior { [ uses-clause ] ... spec-case-seq }
spec-case-seq ::= spec-case [ also spec-case ] ...
spec-case ::= [ model-var-decl ] ...
              [ requires-clause ] ...
              [ measured-clause ] ...
              spec-case-body
              [ example ] ...
spec-case-body ::= { spec-case-seq }
                 | [ when-clause ] ...
                   [ modifiable-clause ] ...
                   [ callable-clause ]
                   ensures-clause [ ensures-clause ] ...
model-var-decl ::= model type-spec pred-variable-declarators ;
requires-clause ::= requires [ redundantly ] pre-cond ;
pre-cond ::= predicate
measured-clause ::= measured [ redundantly ] by pred-expression ;
when-clause ::= when [ redundantly ] predicate ;
modifiable-clause ::= modifiable [ redundantly ] store-ref-list ;
callable-clause ::= callable callable-methods-list ;
callable-methods-list ::= method-name [ , method-name ] ...
                        | everything
                        | nothing
ensures-clause ::= ensures [ redundantly ] [ liberally ] post-cond ;
post-cond ::= predicate
example ::= example [ liberally ] predicate ;
```

Figure 26: Syntax of behavioral specification for methods.

predicate ::= *pred-expression*
pred-expression-list ::= *pred-expression* [**,** *pred-expression*] ...
pred-expression ::= *pred-conditional-expr*
pred-conditional-expr ::= *pred-equivalence-expr*
[**?** *pred-conditional-expr* **:** *pred-conditional-expr*]
pred-equivalence-expr ::= *pred-implies-expr* [**<=>** *pred-implies-expr*] ...
pred-implies-expr ::= *pred-logical-or-expr* [**=>** *pred-implies-expr*]
pred-logical-or-expr ::= *pred-logical-and-expr* ['**||**' *pred-logical-and-expr*] ...
pred-logical-and-expr ::= *pred-inclusive-or-expr* [**&&** *pred-inclusive-or-expr*] ...
pred-inclusive-or-expr ::= *pred-exclusive-or-expr* ['**|**' *pred-exclusive-or-expr*] ...
pred-exclusive-or-expr ::= *pred-and-expr* [**^** *pred-and-expr*] ...
pred-and-expr ::= *pred-equality-expr* [**&** *pred-equality-expr*] ...
pred-equality-expr ::= *pred-relational-expr* [**==** *pred-relational-expr*] ...
| *pred-relational-expr* [**!=** *pred-relational-expr*] ...
pred-relational-expr ::= *pred-shift-expr* **<** *pred-shift-expr*
| *pred-shift-expr* **>** *pred-shift-expr*
| *pred-shift-expr* **<=** *pred-shift-expr*
| *pred-shift-expr* **>=** *pred-shift-expr*
| **informally** *informal-description*
| **informally** *string-literal* [*string-literal*] ...
pred-shift-expr ::= *pred-additive-expr* [*shift-op* pred-additive-expr] ...
shift-op ::= **<<** | **>>** | **>>>**
pred-additive-expr ::= *pred-mult-expr* [*additive-op* *pred-mult-expr*] ...
additive-op ::= **+** | **-**
pred-mult-expr ::= *pred-cast-expr* [*mult-op* *pred-cast-expr*] ...
mult-op ::= ***** | **/** | **%**
pred-cast-expr ::= **(** *type-spec* **)** *pred-cast-expr*
| **+** *pred-cast-expr*
| **-** *pred-cast-expr*
| **~** *pred-cast-expr*
| **!** *pred-cast-expr*
| *pred-postfix-expr* [**instanceof** *type-spec*]
pred-postfix-expr ::= *pred-primary-expr* [*pred-primary-suffix*] ...
pred-primary-suffix ::= **.** *ident*
| **. this**
| **. class**
| '**[**' *pred-expression* '**]**'
| **(** [*pred-expression-list*] **)**

Figure 27: Syntax of predicates and predicate expressions, part 1 of 2.

```
pred-primary-expr ::= ident
                    | builtInType . class
                    | pred-new-expr
                    | constant
                    | super
                    | true
                    | false
                    | this
                    | null
                    | ( pred-expression )
                    | returns
                    | result
                    | throws ( type-spec )
                    | thrown ( type-spec )
                    | unmodified ( store-ref-list )
                    | fresh ( pred-expression-list )
                    | old ( pred-expression )
                    | pred-quantified-expr
pred-new-expr ::= new type pred-new-suffix
pred-new-suffix ::= ( [ pred-expression-list ] ) [ pred-init-block ]
                  | pred-array-decl [ pred-array-initializer ]
                  | { type-spec quantified-var-declarator | predicate }
pred-array-decl ::= pred-dim-exprs [ dims ]
pred-dim-exprs ::= '[' pred-expression ']' [ '[' pred-expression ']' ] ...
dims ::= '[' ']' [ '[' ']' ] ...
pred-init-block ::= { [ pred-init-field-or-semi ] ... }
pred-init-field-or-semi ::= pred-initfield | ;
pred-initfield ::= modifiers type-spec pred-variable-declarators ;
pred-variable-declarators ::= pred-variable-declarator
                              [ , pred-variable-declarator ] ...
pred-variable-declarator ::= ident [ dims ] [ = pred-initializer ]
pred-array-initializer ::= { [ pred-initializer [ , pred-initializer ] ... [ , ] ] }
pred-initializer ::= pred-expression
                   | pred-array-initializer
pred-quantified-expr ::= quantifier ( quantified-vars ) '[' predicate ']'
quantifier ::= forall | exists
quantified-vars ::= type-spec quantified-var-decls
                    [ ; type-spec quantified-var-decls ] ... [ ; ]
quantified-var-decls ::= quantified-var-declarator [ , quantified-var-declarator ] ...
quantified-var-declarator ::= ident [ dims ]
```

Figure 28: Syntax of predicates and predicate expressions, part 2 of 2.

uses-clause ::= `uses` *theory-list* `;`
theory-list ::= *theory-ref* [`,` *theory-ref*] ...
theory-ref ::= *ident* [`(` *renaming* `)`]
renaming ::= *ident*
| *replace-list*
| *ident* `,` *renaming*
replace-list ::= *replace* [`,` *replace*] ...
replace ::= *math-type* `for` *formal*
formal ::= *math-type*
| *ident* `:` *signature*
signature ::= [*math-type-list*] `->` *math-type*
math-type ::= *ident* ['`[`' *math-type-list* '`]`']
math-type-list ::= *math-type* [`,` *math-type*] ...

Figure 29: Syntax of uses clauses.

The nonterminal *java-literal* represents Java literals which are taken without change from Java [7].

A.2.1 White Space

Blanks, horizontal and vertical tabs, carriage returns, formfeeds, and newlines, collectively called *white space*, are ignored except as they serve to separate tokens. Newlines are special in that they cannot appear in some contexts where other whitespace can appear, and are also used to end C++-style (`//`) comments. This is described formally in Figure 34.

A.2.2 Comments

Both kinds of Java comments are allowed in JML (see Figure 35): old C-style comments and new C++-style comments. However, if what looks like a comment starts with the at-sign (`@`) character, then it is considered to be the start of an annotation by JML, and not a comment.

A.2.3 Annotation Markers

If what looks to Java like a comment starts with an at-sign (`@`) as its first character, then it is not considered a comment by JML. We refer to the tokens between `//@` and the following newline, and between pairs of `/*@` and `@*/` as *annotations*. Annotations look like comments to Java, and are thus ignored by it, but they are significant to JML. This is achieved by having JML drop (i.e., do nothing with) the character sequences that are *annotation-markers*: `//@`, `/*@`, and `@*/`. However, JML does recognize certain keywords only within annotations.

Within annotations, an at-sign (`@`) at the beginning of a line is also ignored.

The definition of an annotation marker is given in Figure 36.

statement ::= *compound-statement*
| *variable-decls* `;`
| *ident* `:` *statement*
| *expression* `;`
| `if (` *expression* `)` *statement* [`else` *statement*]
| [*maintaining*] ... [*decreasing*] ... *loop-stmt*
| `break` [*ident*] `;`
| `continue` [*ident*] `;`
| `return` [*expression*] `;`
| *switch-statement*
| *try-block*
| `throw` *expression* `;`
| `synchronized (` *expression* `)` *statement*
| `;`
| `assert` [`redundantly`] *predicate* `;`
| *requires-clause*
| *callable-clause*
| *modifiable-clause*
| *measured-clause*
| *ensures-clause*
| *behavior*

maintaining ::= `maintaining` [`redundantly`] *predicate*
decreasing ::= `decreasing` [`redundantly`] *pred-expression*
loop-stmt ::= `while (` *expression* `)` *statement*
| `do` *statement* `while (` *expression* `) ;`
| `for (` [*for-init*] `;` [*expression*] `;` [*expression-list*] `)` *statement*

for-init ::= *variable-decls*
| *expression-list*

switch-statement ::= `switch (` *expression* `) {` [*switch-body*] ... `}`
switch-body ::= *switch-label-seq* [*statement*] ...
switch-label-seq ::= *switch-label* [*switch-label*] ...
switch-label ::= `case` *expression* `:`
| `default :`

try-block ::= `try` *compound-statement* [*handler*] ...
handler ::= `catch (` *param-declaration* `)` *compound-statement*

Figure 30: Syntax of statements.

expression ::= *assignment-expr*
expression-list ::= *expression* [, *expression*] ...
assignment-expr ::= *conditional-expr* [*assignment-opt assignment-expr*]
assignment-op ::= = | += | -= | *= | /= | %= | >>=
| >>>= | <<= | &= | ' |=' | ^=
conditional-expr ::= *logical-or-expr*
[? *conditional-expr* : *conditional-expr*]
logical-or-expr ::= *logical-and-expr* ['||' *logical-and-expr*] ...
logical-and-expr ::= *inclusive-or-expr* [&& *inclusive-or-expr*] ...
inclusive-or-expr ::= *exclusive-or-expr* ['|' *exclusive-or-expr*] ...
exclusive-or-expr ::= *and-expr* [^ *and-expr*] ...
and-expr ::= *equality-expr* [& *equality-expr*] ...
equality-expr ::= *relational-expr* [== *relational-expr*] ...
| *relational-expr* [!= *relational-expr*] ...
relational-expr ::= *shift-expr* < *shift-expr*
| *shift-expr* > *shift-expr*
| *shift-expr* <= *shift-expr*
| *shift-expr* >= *shift-expr*
| informally *string-literal* [*string-literal*] ...
shift-expr ::= *additive-expr* [*shift-op* additive-expr] ...
shift-op ::= << | >> | >>>
additive-expr ::= *mult-expr* [*additive-op mult-expr*] ...
additive-op ::= + | -
mult-expr ::= *cast-expr* [*mult-op cast-expr*] ...
mult-op ::= * | / | %
cast-expr ::= (*type-spec*) *cast-expr*
| ++ *cast-expr*
| -- *cast-expr*
| + *cast-expr*
| - *cast-expr*
| ~ *cast-expr*
| ! *cast-expr*
| *postfix-expr* [instanceof *type-spec*]
postfix-expr ::= *primary-expr* [*primary-suffix*] ...
primary-suffix ::= . *ident*
| . this
| . class
| '[' *expression* ']'
| ([*expression-list*])
| ++
| --

Figure 31: Syntax of expressions, part 1 of 2.

```
primary-expr ::= ident
               | builtInType . class
               | new-expr
               | constant
               | super
               | true
               | false
               | this
               | null
               | ( expression )
builtInType ::= void
              | boolean
              | byte
              | char
              | short
              | int
              | long
              | float
              | double
constant ::=
new-expr ::= new type new-suffix
new-suffix ::= ( [ expression-list ] ) [ class-block ]
             | array-decl [ array-initializer ]
array-decl ::= dim-exprs [ dims ]
dim-exprs ::= '[' expression ']' [ '[' expression ']' ] ...
dims ::= '[' ']' [ '[' ']' ] ...
array-initializer ::= { [ initializer [ , initializer ] ... [ , ] ] }
initializer ::= expression
              | array-initializer
```

Figure 32: Syntax of expressions, part 2 of 2.

```
microsyntax ::= lexeme [ lexeme ] ...
lexeme ::= white-space | comment | annotation-marker | token
token ::= ident | keyword | special-symbol | java-literal | informal-description
```

Figure 33: Microsyntax of JML.

white-space ::= *non-nl-white-space* | *newline*
non-nl-white-space ::= a blank, tab, carriage return, or formfeed character
newline ::= a newline character

Figure 34: Microsyntax of white space.

comment ::= *C-style-comment* | *C++-style-comment*
C-style-comment ::= `/*` [*C-style-body*] *C-style-end*
C-style-body ::= *non-at-star* [*non-star-slash*] ...
| stars-non-slash [non-star-slash] ...
non-star-slash ::= *non-star*
| *stars-non-slash*
stars-non-slash ::= `*` [`*`] ... *non-slash*
non-at-star ::= any character except `@` or `*`
non-star ::= any character except `*`
non-slash ::= any character except `/`
C-style-end ::= [`*`] ... `*/`
C++-style-comment ::= `//` *newline*
| `//` *non-at-newline* [*non-newline*] ... *newline*
non-newline ::= any character except a newline
non-at-newline ::= any character except `@` or newline

Figure 35: Microsyntax of comments.

annotation-marker ::= `//@` | `/*@` | `@*/`
ignored-at-in-annotation ::= `@`

Figure 36: Microsyntax of annotation markers.

ident ::= *letter* [*letter-or-digit*] ...
letter ::= `_`, `$`, `a` through `z`, or `A` through `Z`
digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
letter-or-digit ::= *letter* | *digit*

Figure 37: Microsyntax of tokens

keyword ::= *java-keyword* | *jml-keyword*
jml-keyword ::= `also` | `assert` | `behavior`
| `by` | `callable` | `constraint`
| `decreasing` | `depends` | `ensures`
| `everything` | `example` | `exists`
| `forall` | `fresh` | `informally`
| `initially` | `invariant` | `liberally`
| `maintaining` | `measured` | `model`
| `modifiable` | `nothing` | `old`
| `on` | `pure` | `reach`
| `redundantly` | `refine` | `represents`
| `requires` | `result` | `returns`
| `thrown` | `unmodified` | `uses`
| `weakly` | `when`

Figure 38: Microsyntax of keywords.

A.2.4 Tokens

Character strings that are Java keywords are made into the token for that keyword, instead of being made into an *ident* token. Within an *annotation* this also applies to JML keywords. The details are in Figure 37.

Several strings that would otherwise be *ident*s are reserved as keywords. Java keywords are recognized in all contexts, but JML keywords are only recognized as such within annotations. See Figure 38. The nonterminal *java-keywords* represents the keywords in Java 1.1.

The nonterminal *java-special-symbol* is the special symbols of Java, taken without change from Java [7]. See Figure 39.

An *informal-description* looks like `(* some text *)`. It is used in predicates following the keyword `informally`. See Figure 40.

special-symbol ::= *java-special-symbol* | *jml-special-symbol*
jml-special-symbol ::= `=>` | `<=>`

Figure 39: Microsyntax of special symbols.

informal-description ::= `(*` *non-star-close* [*non-star-close*] ... `*)`
non-star-close ::= *non-star*
| *stars-non-close*
stars-non-close ::= `*` [`*`] ... *non-close*
non-close ::= any character except `)`

Figure 40: Microsyntax of informal descriptions.